

Multi-Centroid PSO Classification Learning on the GPU

Cain Cresswell-Miley

`cjc167@uclive.ac.nz`

**Department of Computer Science and Software Engineering
University of Canterbury, Christchurch, New Zealand**

Supervisor: Kourosh Neshatian

Abstract

Training classifiers can be seen as an optimization problem. With this view, we have developed a method to train a type of nearest centroid classifier with PSO. Results showed an improvement on most of the datasets tested. Additionally, we have developed a method to utilize the developed classifier with datasets containing both numeric and categorical data by integrating the centroid algorithm with a decision tree. However, experiments found no significant improvement over the original decision tree method. Both the developed PSO centroid algorithm, and the previous PSO centroid algorithm are implemented on the GPU, with results showing at least one order of magnitude difference between speeds of the GPU and a ‘typical’ sequential CPU implementation.

Contents

1	Introduction	2
1.1	Motivation	3
1.2	Goal	3
1.3	Report Outline	3
2	Background	4
2.1	Particle Swarm Optimization	4
2.2	CUDA - GPU Programming	5
2.3	Classification	7
2.4	Decision tree	8
2.5	Related Work	10
2.5.1	PSO for Classification	10
2.5.2	PSO on the GPU	11
3	Design	12
3.1	Classifier Representation	12
3.2	Stepwise Centroid Algorithm	13
3.3	GPU PSO	16
3.4	GPU Stepwise Centroid Implementation	16
3.5	Categorical Data	18
4	Experiment Design	21
4.1	GPU implementation	21
4.2	Stepwise Centroid algorithm	22
4.3	MADS and SADS	22
4.4	Datasets	22
4.5	Configuration settings	23
5	Results and Discussion	25
5.1	GPU Implementation	25
5.2	Stepwise Centroid Accuracy	28
5.2.1	Mixed datasets	29
6	Conclusion	33
6.1	Limitations and Future Work	34

1

Introduction

In machine learning, the area of supervised learning focuses on the task of training classifiers by learning from a set of labelled training examples, with the goal of accurately classifying future data. Generally, training classifiers is an optimization problem. That is, learning from a set of training examples can be seen as optimizing a given performance metric (objective), such as the misclassification rate over the training data, by modifying the models parameters or structure.

Classification problems can contain different types of data. Broadly speaking, individual examples may consist of numeric or categorical features. Different classification algorithms handle different types of data. In the continuous space, classifiers are typically considered to be defining decision boundaries in a continuous space, whereas for categorical data, classifiers are typically in the logical space.

Particle Swarm Optimization (PSO) is a biologically-inspired optimization technique based on simulating the behaviour of swarms, such as schools of fish or bird flocks. Conceptually, the algorithm works by iteratively moving a population of ‘particles’ around a search space, where each particle’s direction is influenced by the particle in its local neighbourhood that maximizes the chosen objective function for the problem [21].

The GPU is a processor specialized for highly-parallel and compute-intensive tasks. Since the introduction of the general purpose GPU platforms, CUDA and OpenCL, using the GPU for non graphics-related tasks has become a popular research area. In order to achieve high-parallelism, the GPU devotes more transistors to data processing rather than caching and flow control. This trade-off limits the types of algorithms that may be effectively implemented on the GPU, and requires algorithms be specifically designed with these trade-offs in mind [19].

In this research, we apply PSO to learning classifiers in the continuous space, by implementing a classifier learning algorithm for a nearest centroid based classifier representation. Additionally, we implement the algorithm on the GPU and evaluate the performance compared to an equivalent CPU implementation to gauge the effectiveness of utilizing the GPU for classification using PSO. We also further extend the proposed classifier for use with mixed data sets that contain both numeric, and categorical data.

1.1 Motivation

Classification can be a hard problem, and different classifier models are capable of representing different hypotheses and concepts. Using evolutionary computing algorithms such as PSO to learn classifiers can give more freedom in the types of classifier models that can be effectively learned, since there are no requirements on the structure of the problem. These algorithms are simple optimization algorithms that attempt to find globally optimal solutions through a biased random search, and have been applied successfully in many different areas [20]. However, evolutionary computing methods, and PSO in particular can suffer from slow convergence for large scale problems. Therefore, we aim to implement a PSO approach to classification on the GPU in order to evaluate the algorithm's effectiveness for classification tasks, and the feasibility and advantages of a PSO implementation on the GPU.

1.2 Goal

The aim of this research is to investigate the use of PSO for classification tasks, and investigate effectively implementing PSO on the GPU for classification tasks, where the optimization objective is evaluated across a number of different sized data sets.

The research questions are as follows:

- What type of classifier representation and corresponding optimization problems are most appropriate for PSO?
- How can PSO be effectively utilized for classification?
- Is a GPU implementation more effective than a CPU implementation?
- How does a GPU implementation of PSO for classification scale with the size of the data?

1.3 Report Outline

The report begins with an overview of the background and previous work related to this research. We introduce the basic background topics related to the research, and then give an overview of the research done related to this work. Chapter 3 then outlines and describes the implementation and design of the methods we have developed. Chapter 4 describes how the experiments have been designed to evaluate different aspects of the developed algorithms. Chapter 5 describes the results of the experiments, with some discussion and analysis. Finally, Chapter 6 gives the conclusions and future work from this research.

2

Background

This chapter contains an introduction to the concepts utilized in this research, and a literature review of related work. We cover the information necessary to contextualize future chapters. In particular, we introduce Particle Swarm Optimization, GPU programming, Decision Trees, and a review of the work related to our topic.

2.1 Particle Swarm Optimization

Particle Swarm Optimization is a population based optimization technique introduced by Kennedy and Eberhart [11]. PSO is simple to implement, and does not require any derivative information, or any additional problem structure other than an objective function. Because of these properties, it has been applied to many different tasks, such as Floor planning [24], and Reactive power control [3], and training of classifiers [23].

To perform optimization, A population of particles (a swarm) is iteratively moved through a multi-dimensional continuous search space until a sufficiently ‘fit’ solution is attained (or a maximum number of iterations is reached). Each particle is defined by a multi-dimensional vector p , representing the position. During search, each particle has an associated velocity, controlling how the particle moves around. At each iteration the position and velocity of each particle is updated according to the following mathematical equations, where i is the current iteration index: [21]

$$\begin{aligned}v_{i+1} &= w\mathbf{v}_i + c_1\mathbf{r}_1 \otimes (\mathbf{p}_i^{\text{best}} - \mathbf{p}_i) + c_2\mathbf{r}_2 \otimes (\mathbf{p}_i^{\text{nbest}} - \mathbf{p}_i) \\p_{i+1} &= p_i + v_{i+1}\end{aligned}$$

The velocity is updated such that the previous velocity, the best known position of the current particle so far, \mathbf{p}^{best} , and the best known position across the particles neighbourhood, $\mathbf{p}^{\text{nbest}}$, all influence the particle’s movement. w is known as the inertia weight, which controls the influence of the previous velocity, and c_1 and c_2 control the influence of the current particle’s last best position and the neighbourhoods best known position respectively. Finally, \mathbf{r}_1 and \mathbf{r}_2 are vectors of random numbers uniformly distributed in the interval $[0, 1]$, and are intended to introduce randomness into the search. The operator \otimes performs component-wise multiplication of two vectors.

Algorithm 1: Basic PSO Pseudocode

```

1 Initialize the position and velocity of all particles;
2 while max iterations not reached do
3   calculate fitness of each particle;
4   update each particle's personal and neighbourhood best;
5   update each particle's position and velocity;
6 end

```

The neighbourhood of a particle is defined topologically—by viewing each particle as a node in a graph, a particle's neighbourhood is the set of particles that it is connected with. In the *gbest* topology, the particles form a fully connected graph, and thus every particle influences each other. Conversely, for *lbest* topologies, each particle is influenced by a strict subset of particles in the whole population. A popular *lbest* topology is the ring topology, where each particle has two distinct neighbours [12]. The way the neighbourhood is defined affects how the best position is communicated throughout the swarm. This appears to affect the convergence properties of the algorithm, with experiments showing that a ring-like topology appears to be better at exploration, whereas the star topology converges faster (although not necessarily to a globally optimal result) [25].

Algorithm 1 gives the basic outline of the PSO algorithm as described above. The fitness calculation is defined by the application utilizing PSO, however the other steps are independent of the problem.

2.2 CUDA - GPU Programming

In 2007, NVIDIA introduced CUDA, a free, proprietary, parallel computation platform and programming model. Before this, general purpose computation on the GPU required interacting with the GPU through graphical APIs such as OpenGL and DirectX, and therefore required representing the problem as operations on graphics primitives. Since this introduction of CUDA, general purpose GPU computing has become a popular area of research for implementing many different computational techniques. For example, general purpose GPU computing with CUDA has been applied to parallelizing regular expression pattern matching [4], DNA sequence alignment using a suffix tree based method [26], and accelerating graph algorithms over large graphs, such as All Pairs Shortest Path APSP, or Breadth First Search [9].

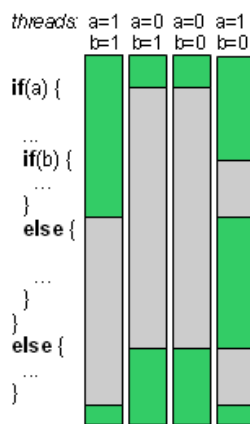
CUDA is a parallel computing platform and programming model developed by NVIDIA for general purpose computing on the GPU. The GPU is specifically designed for data parallel algorithms that can take advantage of a high degree of parallelism without a high degree of reliance on flow control or memory caching. More specifically, the GPU is well

suited to programs which can be specified in terms of multiple threads executing the same code across multiple different pieces of data in parallel [19].

In the CUDA programming model, code is executed across a number of parallel threads, where each thread is distributed among a number of thread blocks. Each thread block is mapped to a single multiprocessor on the GPU, which allows threads that share a thread block to have the ability to synchronize with each other, and access shared memory, but separate thread blocks are executed independently from each other.

Within this model, coding for the GPU consists of writing kernels, which are special C functions written to be executed in parallel on the GPU. The kernel is executed across a number of thread blocks with each thread block consisting of a chosen number of threads. The kernel has access to the runtime constants *threadIdx*, *blockIdx*, *blockDim*, *gridDim*, specifying information about the current thread executing the kernel. This gives the ability to map different pieces of data to different threads, and have a more fine grained control over the execution path of each thread.

Figure 2.1: Diagram describing how threads are executed within a warp. Green means the thread is executing, whereas grey means the threads are inactive [14]



A thread block is executed by a GPU's multiprocessor by first partitioning the threads into groups of 32 parallel threads, called *warps*, and then scheduling each warp for execution. A warp executes one common instruction at a time for every thread, so if threads in a warp diverge via data-dependent conditional branching, the warp executes each branch serially, disabling threads which are not on that path, and converges back to a single path once each branch has been executed. This is illustrated in Figure 2.1. Full efficiency in a warp is therefore achieved when all 32 threads agree on the same execution path.

Threads have access to multiple memory spaces during execution. Figure 2.2 illustrates the layout of the memory spaces. Thread local memory and global memory (along with the constant and texture memory spaces) are both stored on-device, whereas registers and shared memory are both stored on-chip (the multiprocessor). Shared memory and the

registers therefore have much higher bandwidth and lower latency than local and global memory, but are not as large as the global or local memory spaces.

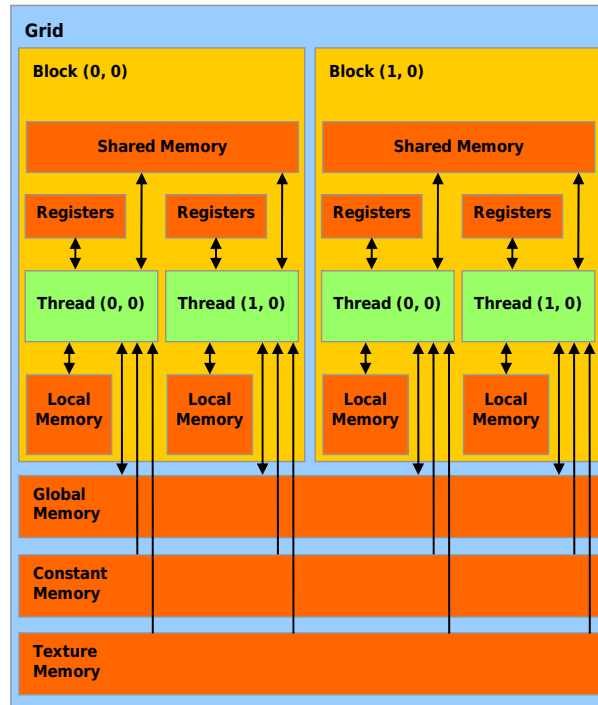


Figure 2.2: CUDA Memory Hierarchy [19]

2.3 Classification

Learning classifiers is a task in the field of supervised learning. Classifier learning algorithms learn (or induce) a classifier by attempting to optimize the structure, or parameters of a classifier over a set of pre-labelled examples, known as a training set. An example is a vector of feature values, where features can be of multiple different types. These types can be broadly categorized as numeric or categorical. Different classifiers and classifier learning algorithms are specifically designed with respect to the different feature types, with some classifiers only handling one type of feature.

Bias of a classifier or classifier learning algorithm refers to properties that impose a sort of structure to the classifier. Bias is a necessary component of classifier learning, in order to ensure a classifier can generalize to classify future examples. Without sufficient bias, a classifier can overfit the training data, since the model can continually be modified, and made more complex until the training data is fit perfectly. The extreme case is a complete memorization of the training set. An example of classifier bias would be limiting a classifier to a linear representation, or imposing a limit on the number of hidden nodes in a neural network. A validation set can be used during training of a classifier within

certain algorithms in order to allow a classifier to be sufficiently complex, yet still capable of generalizing to unseen examples. The validation set is a partition of the training data that is set aside in order to monitor, or evaluate the complexity of the model being learnt during training.

2.4 Decision tree

Decision trees are a popular type of symbolic classifier that is capable of utilizing both numeric and categorical data for classification. A decision tree is built of test nodes, and decision nodes at the leaves. A test node examines a feature, or features, and branches based on the on the possible values. The decision nodes are assigned a particular class from the problem, and are used to perform the actual classification of an example, by assigning a class to examples which reach the leaf. That is, a particular path from the root of the tree, to a leaf node can be seen as a conjunction of conditions required for an example to be assigned a particular class by a decision node.

The most common classification learning algorithm, C4.5 [22], is capable of producing tests for both categorical, and numeric data. For categorical features, a node of a decision tree has a separate branch for each possible value of the feature. For numeric features, C4.5 only splits on a single feature at a time, and branches based on whether a numeric feature is greater, or less than, some threshold chosen to maximize the perceived split between the classes.

C4.5 is a greedy, top-down decision tree construction algorithm. A test is selected according to some performance metric on the training data, where the performance metric measures how well a particular feature discriminates between the classes in the problem. Once a test is selected, each subtree is recursively built on the subsets corresponding to each subtree. The algorithm terminates once the examples in the subtree are of a single class, or there are zero examples belonging to the current subtree. In the former case, the subtree will be made a leaf that identifies the class of the examples. In the latter case, a leaf is created, and assigned the most common class of the subtrees parents.

The performance metric used for test selection in the C4.5 algorithm is known as gain ratio, which is a modification of information gain. Each of these are a measure for the average reduction in uncertainty with respect to the output classes that is achieved by selecting a given test.

The information conveyed by a message or event is defined as $-\log_2(P)$ bits, where P is the probability of the event. The idea is, events with a lower probability convey more information, whereas higher probabilities contain less information. Let P_j be the probability

of selecting an instance from the training set T with class C_j , then

$$P_j = \frac{|\{x : x \in T, \text{class}(x) = C_j\}|}{|T|}$$

Hence, the amount of information conveyed by a single example with class C_j is:

$$-\log_2(P_j)$$

Now, the average amount of information necessary to identify the class of an instance in T , or the *entropy*, is calculated as:

$$\text{info}(T) = \mathbf{E}[-\log_2(X)] = -\sum_{j=1}^k P_j \times \log_2(P_j)$$

Once the training set T has been partitioned into n separate subsets $\{T_1, T_2, \dots, T_n\}$ by a test Y , the expected class information in each partition is calculated as follows:

$$\text{info}_Y(T) = \sum_{i=1}^n \frac{|T_i|}{|T|} \times \text{info}(T_i)$$

The *gain* of information from partitioning with test Y is then calculated as:

$$\text{gain}(T) = \text{info}(T) - \text{info}_Y(T)$$

This measures the gain of information from partitioning T according to the test Y .

However, Quinlan [22] noted that the gain criterion has a strong bias towards tests with many outcomes. The gain ratio criterion alleviates this by scaling the information gain according to the number of outcomes of the test. The specific calculation we use, is known as the split information gain, which is calculated as follows:

$$\text{split info}(X) = -\sum_{i=1}^n \frac{|T_i|}{|T|} \times \log_2\left(\frac{|T_i|}{|T|}\right)$$

We then divide the *information gain*, which measures the gain of information pertaining to classification generated by the same partitions by the *split info*:

$$\text{gain ratio}(X) = \frac{\text{gain}(X)}{\text{split info}(X)}$$

This expresses the proportion of information generated by the split that appears helpful for classification.

2.5 Related Work

2.5.1 PSO for Classification

Applying PSO to classification has been achieved in multiple ways, such as using PSO to train neural networks [8, 7, 10], learning sets of hierarchical classification rules [23], or finding centroids in the feature space [5].

Sousa *et al.* [23] introduce a rule based approach for classifying categorical data using PSO. A hierarchical set of *if-then* rules are created, where the condition of each rule is a conjunction of feature values. An example meets the condition if each relevant feature has the same feature value. The set of rules is created by iteratively discovering new rules that classify the predominant class in the current data set, with the current data set being updated in each iteration by removing examples that match the newly created rule. Binary PSO is used as the underlying algorithm to find the conjunction of feature values that most optimally matches examples of the given class. While the above paper considered problems consisting of categorical features, Falco *et al.* [5] describe a classification algorithm using PSO for continuous data sets. For a classification problem with C output classes, a classifier is represented by C centroids in the feature space, where each centroid is mapped to a single output class. An example is assigned a class by the closest centroid (in terms of Euclidean distance). Given this representation, PSO is used to optimize the position of the centroids, such that the accuracy over the training set is maximized. However, by using only one centroid per class, this classifier representation is limited to representing linear boundaries between each class, which limits the types of data this classifier can accurately classify.

Mohammed and Zhang [16] evaluate the performance of a single centroid PSO classifier, as described above, using different distance metrics. The algorithm with different distance metrics was compared to a basic nearest centroid classifier, and two nearest neighbour classification algorithms. The basic nearest centroid classifier simply utilizes the mean point of each class as the centroid position for the class. The nearest neighbour algorithms are similar, but use a subset of training examples to define what is equivalent to a centroid position. The Euclidean distance measure was compared to the class dependent Mahalanobis (CMD) distance, which takes into account how points in a single class diverge from the mean point for that class. Experiments found that while the Euclidean distance measure was superior to the simple centroid classifier, and the nearest neighbour classifiers, but was only better than the same algorithm using CMD distance on two datasets. The authors however argued that a motivation for not using the simple Euclidean distance is that it assumes that the input space is homogeneous. However, for classification, if features are scaled into a fixed range of values, the homogeneity assumption should not be problematic to the performance of a centroid based classifier.

2.5.2 PSO on the GPU

While PSO has shown some success in its use for classification, convergence of the algorithm can be slow. By taking advantage of the GPU, a processor designed for massively-parallel computations, PSO may be more useful. The GPU is a processor specialized for highly-parallel and compute intensive tasks, and since the introduction of general GPU computing platforms such as CUDA and OpenCL utilizing the GPU to parallelize algorithms has become a popular research area. In order to achieve high parallelism, the GPU devotes more transistors to data processing rather than caching and flow control. This tradeoff limits the types of algorithms that can benefit from the GPU, and requires that algorithms be specifically designed with these tradeoffs in mind.

Two methods of implementing PSO for the GPU have been described in the literature. Zhou and Tan [28] implement PSO by parallelizing each individual stage of the PSO algorithm. This approach scaled well to more particles and larger dimensions, although each stage of the algorithm required reading particle data from the GPU's global memory, which has high latency. Mussi *et al.* [18] implemented PSO as a single GPU function, where each thread on the GPU performed a single step of the algorithm for a single particle. However, due to synchronisation limitations of the GPU, a swarm was limited to a single thread block, and hence a single GPU multiprocessor, which limits the utilisation of the GPU if only a single swarm is being used. Additionally, because each particle is mapped to a single thread, the number of particles in a swarm is limited by the maximum number of threads in a thread block.

3

Design

In this chapter we introduce the design of a nearest centroid classification algorithm using multiple centroids per class, and describe a stepwise learning algorithm based on PSO for learning such a classifier. Additionally, we describe a GPU implementation of the designed algorithm and extend the algorithm for use on mixed datasets by integrating it with a decision tree classifier.

3.1 Classifier Representation

In order to use PSO for classification, we require a continuous representation of a classifier. Therefore, we have developed multi centroid classifier representation, in order to allow the centroids to define more complex decision boundaries. That is, a classifier is represented by a set of centroids, K , in the feature space. Each centroid $\kappa_i \in K$ is assigned a class $c_j \in C$ by the surjective map *class*, where $|K| \geq |C|$. Let $d(a, \kappa)$ define the distance between an example u and the centroid κ , and κ^* be the centroid closest to an example, then we have:

$$\begin{aligned} C &= \{c_1, c_2, \dots, c_{|C|}\} \\ K &= \{\kappa_1, \kappa_2, \dots, \kappa_{|K|}\} \\ \textit{class}: K &\mapsto C \\ \kappa^* &= \arg \min_{\kappa \in K} d(u, \kappa) \\ h(u) &= \textit{class}(\kappa^*) \end{aligned}$$

The function $h(u)$ defines how an example is classified given the classifier represented by K and the map *class*. In particular, an example u is assigned the class of the centroid κ^* , which is the closest centroid to the given example.

By allowing multiple centroids to map to a single class, the classifier is capable of representing complex decision boundaries between individual classes of a problem. Because examples are classified by the closest centroid in feature space, the decision boundaries for such a classifier can be considered piecewise linear. Figure 3.1 illustrates how these boundaries are defined. Specifically, a boundary exists between two centroids of different classes if there exists a point between the two centroids that is equally close to both centroids,

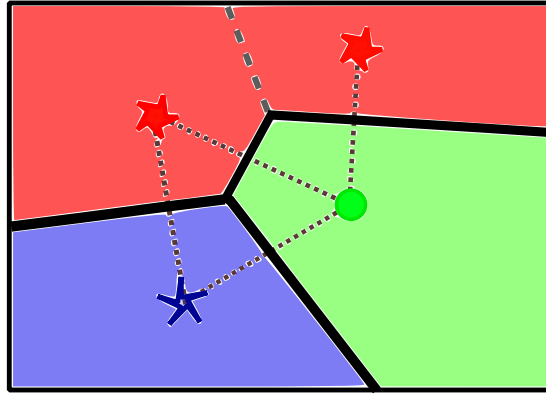


Figure 3.1: Decision boundaries defined by four centroids together representing three different classes.

and is not closer to any other centroid. Such a boundary is defined as the line where all points are equidistant to both centroids.

Figure 3.2 displays two different classification problems. The linear function can be solved with a single centroid per class, which allows the classifier to define a single line decision boundary that separates each class. However, the XOR problem requires more than a single single centroid per class, since XOR problem cannot be fully separated with just a single line. However, by introducing an extra centroid for each class, each of the four regions of the XOR function can be covered. Where the centroids together define the four boundaries required to successfully represent the XOR function.

3.2 Stepwise Centroid Algorithm

In order to learn a nearest centroid classifier with a single centroid per class, a simple optimization algorithm may be used such as PSO, since the structure of the classifier is fixed, just the positions of the centroids must be optimized. However, learning a multi-centroid nearest centroid classifier is more complicated. Because more than one centroid can be assigned to a single class, an algorithm must be designed which can also learn the optimal number of centroids per class in addition to learning the positions of each centroid.

To learn such a classifier, we have developed an iterative algorithm which progressively adds more centroids to an initial classifier, until some stopping condition is reached. More specifically, we initially create a nearest centroid classifier using a single centroid per class, and optimize the positions using PSO. We then continuously add new centroids to the classifier, with the class of the centroid assigned randomly, and then reoptimize the positions of every centroid in the classifier. This continues until some stopping condition is

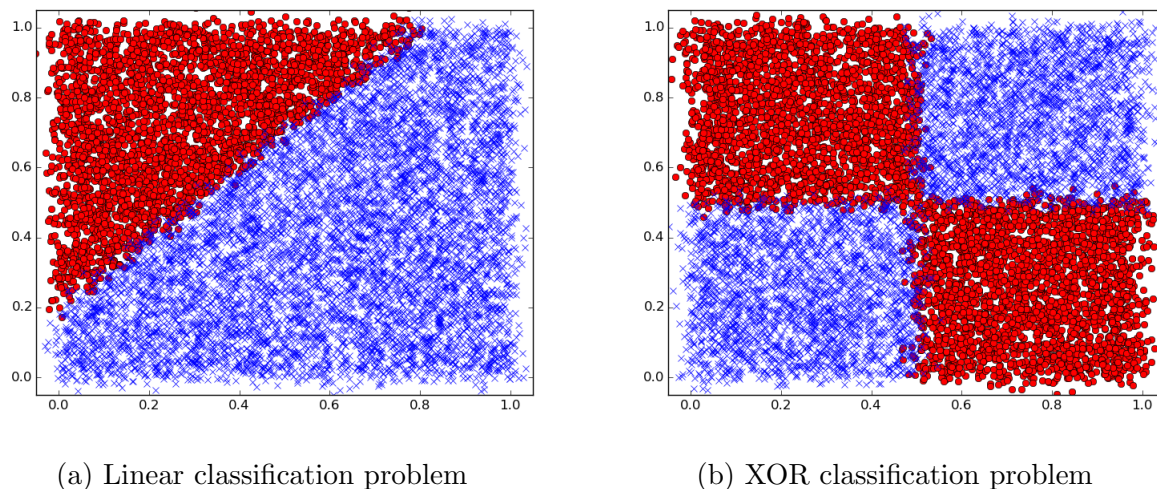


Figure 3.2: Two example classification problems with different, each with two classes (red and black) and two features (dimensions).

met. The class of the new centroid at each step is assigned randomly, rather than to the least-discriminated class, because it was found that assigning the centroid to the least-discriminated class introduced some unwanted bias in the classifiers the algorithm was capable of learning. More specifically, in some cases, adding a new centroid for the least discriminated class was not capable of increasing accuracy of the classifier for that class while also increasing the overall accuracy of the classifier. Algorithm (2) gives an outline of the designed algorithm. Step (1) scales the training data such that every example is contained within a hypercube with lower and upper bounds of 0 and 1. This is performed in order to ensure the distance measure used for finding the closest centroid is not biased by the scaling of different features. Additionally, this also bounds the search space over which PSO finds the positions of each centroid. The data is then partitioned into a training, and validation set, where the validation set is used for evaluating the generalizability of the classifier during the execution of the algorithm.

To begin, Step (3) initializes a nearest centroid classifier with a single centroid for each class by optimizing the centroid positions with PSO. The position of a particle represents the positions of every centroid in a classifier, and the fitness of a particle is defined as the accuracy of the classifier it represents.

New centroids are iteratively added to the classifier with a randomly chosen class. Once a new centroid is added in step (7), PSO reoptimizes the position of every centroid in the classifier, where previously found centroids are initialized to their previous positions. This is performed to allow the boundaries within the feature space, as defined by the centroids, to potentially adapt given the additional centroid. The outer while loop continues until the accuracy (fitness) over the validation set has not increased for a number of iterations n ,

Algorithm 2: Stepwise Centroid

```

1 Scale features of the training data to the range  $[0, 1]$ ;
2 Partition training data into a training set (66%), and validation set;
3 Optimize position of  $|C|$  centroids, one for each class, by using PSO;
4 Calculate accuracy of classifier over validation set;
5 set  $i$  to 0;
6 while  $i < n$  do
7   Add new centroid for a randomly chosen class;
8   Reoptimize positions of all centroids with current centroid added;
9   Calculate accuracy of classifier over validation set;
10  if validation accuracy increased then
11    set  $i$  to 0;
12    store currently learned classifier as the current best;
13  end
14  else
15    set  $i = i + 1$ ;
16  end
17 end
18 Return the current best classifier

```

and once complete, the classifier found throughout the process with the highest accuracy is returned. n is a parameter controlling ‘overstep’. That is, n is used to allow the algorithm to overstep the optimal, and backtrack if necessary. This overstep is included in order to allow for the cases where adding a single centroid does not increase the overall accuracy of the classifier, but adding another centroid will.

The fitness of a particle, or the accuracy of the classifier it represents, is calculated by iterating over every example, finding the closest centroid defined by the particle, and then summing the total number of examples correctly classified by the closest centroid. This total is then divided by the total number of examples to give the classifiers accuracy.

The time complexity of the single centroid PSO algorithm, with the number of PSO iterations and particles held constant is $O(e|C|f)$, where e is the number of examples, $|C|$ is the number of classes, and f is the number of features. That is, for every particle, the algorithm must iterate through all e examples and compare each example to $|C|$ class centroids by iterating over f features. The runtime of the stepwise centroid algorithm however, is sensitive to the complexity of the classification problem at hand. The single centroid PSO algorithm can be considered a component of the stepwise centroid, and therefore the complexity is at least $O(e|C|f)$. In other words, because the algorithm only stops once the error on the validation set has not decreased for a number of iterations n , the overall runtime fluctuates depending on the problem.

3.3 GPU PSO

PSO is a fundamentally sequential algorithm, where each step depends on the previous steps to have been completed, as in Algorithm 1. To parallelize this, the GPU is used as a coprocessor. That is, each individual step of the algorithm is implemented to be ran in parallel on the GPU, whereas the CPU manages the overall control flow of the algorithm by scheduling when to run each individual step on the GPU. The approach outlined here follows the same structure as Zhou and Tan [28]. The other potential solution to parallelizing PSO on the GPU is to implement the whole algorithm as a single kernel on the GPU. However, because communication between thread blocks is not possible on the GPU, no more than one thread block could be used for a single swarm, because of the particle neighbourhood being completely connected.

Each step of the generic PSO algorithm, the initialisation, position and velocity updating, and updating of each particles neighbourhood best, ignoring the fitness calculation, all perform operations on single dimensions of a particle at one time, and therefore each step can be mapped to the GPU architecture in the same manner.

Each particle is mapped to B thread blocks, with X threads per block, where $B \geq 1$ such that $B \times X$ is greater than or equal to the number of dimensions of the particle. In this way, an individual thread is mapped to a single dimension of a single particle, where the *block id* of the thread block identifies the particle, and the *thread id* identifies the particle dimension mapped to the current thread. This defines a surjective mapping from the particles to GPU thread blocks, where more than one thread block is used if the dimension of a particle exceeds the number of threads per block. This allows the algorithm to scale to perform more parallel computations as the number of dimensions of a particle increases.

Each thread block maps to a single particle, which means the position of a single particle may be stored in *shared* memory. This means individual threads in a thread block may read the particle position data from the lower latency shared memory, rather than global memory. Which should lead to higher efficiency.

Currently, the performance of single precision arithmetic on the GPU far outweighs the GPUs performance with double-precision arithmetic. Because of this, PSO has been implemented to use single precision floating point arithmetic. This trades (arithmetic) accuracy for speed. This should have limited effect on the effectiveness of the algorithm, assuming the magnitudes of values used in PSO are approximately equal.

3.4 GPU Stepwise Centroid Implementation

The overall structure GPU implementation of the stepwise centroid is very similar to the described structure. The GPU is used as a coprocessor in order to parallelize individual

steps, while the CPU is still used for the scheduling and overall control flow. In particular, the GPU PSO implementation is used for optimization, and the fitness function is also implemented on the GPU. In general, the fitness function is the most costly aspect to evaluate during optimization, and in the case of applying the algorithm to classification, the runtime of the fitness function is particularly important, in terms of execution time of the algorithm. This is because the fitness function must iterate over every example in the training set in order to calculate the accuracy of the current particle position, and this must be done for every particle on every iteration of the algorithm.

To parallelize the fitness function, each particle is mapped to one or more thread blocks, where each thread in the thread block is mapped to a single example in the training set. If the number of examples exceeds the number of threads per block, additional thread blocks are used for each particle. This allows for the utilization of the GPU to scale as the number of examples in the training set increases.

Every thread performs the task of classifying the assigned example, given the classifier of the particle that is assigned to the thread block of the current thread. To perform the classification, each thread iterates over each centroid defined by the particle, using Euclidean distance as the distance measure. If the class of the closest centroid (the assigned classification of the example) matches with the known class of the example, a value of 1 is written to an array stored in shared memory to the location specified by the current threads ID, and 0 otherwise.

Once the classification of each example is complete, each thread block has a shared array with each element indicating whether or not the corresponding example was classified correctly. To find the total number of correctly classified examples within the thread block, the sum of the elements in the array must be calculated. The trivial method of performing this calculation is for a single thread to iterate over every element in the array and add the value of the element to a counter. However, because multiple threads are executed in parallel within a thread block, such a calculation would require every other thread to simply idle until that single thread has completed the work. This can be made more efficient by implementing a *parallel reduction*, which is capable of utilizing more threads in order to perform the sum operation, by using the associativity of the add operation to reorder the steps so they can be performed in a more parallel way. Figure 3.3 describes this concept more clearly. In the left diagram, each operation on individual levels of the tree may be executed across multiple different threads, whereas in the right diagram each operation depends on the output of a previous operation, and therefore cannot be parallelized as easily. To be specific, the parallel reduction algorithm is implemented by making each thread in the first half of the block add the value corresponding to a thread on the second half of the thread block to its corresponding value in the shared array, and continually doing this operation, halving the number of executing threads each time.

Finally, Once each thread block has summed the total number of correctly classified ex-

amples in the block, the result is divided by the total number of examples in the training set, and is written to a memory location in global memory for the current particle. This operation requires the use of an `atomicAdd` operation, to ensure serialized memory access across multiple thread blocks, since a single particle may be mapped to multiple thread blocks.

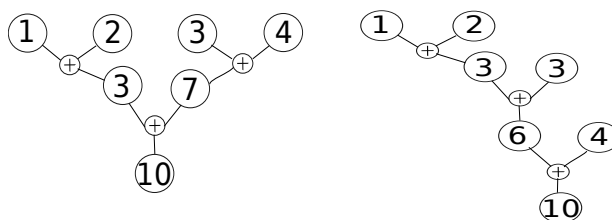


Figure 3.3: On the left is an order of operations allowing more parallelism, on the right, a more serial ordering of operations

3.5 Categorical Data

Classification problems come in many different forms. The types of features present in a problem depends highly on the application area. Datasets containing both categorical and numeric data pose an interesting challenge when designing classifiers. Approaches to learning classifiers for categorical data typically learn logical expressions in some form, whereas classifier learning algorithms for numeric data typically learn decision boundaries between classes within feature space. The challenge is to merge these two views of classification together in order to handle both types of data.

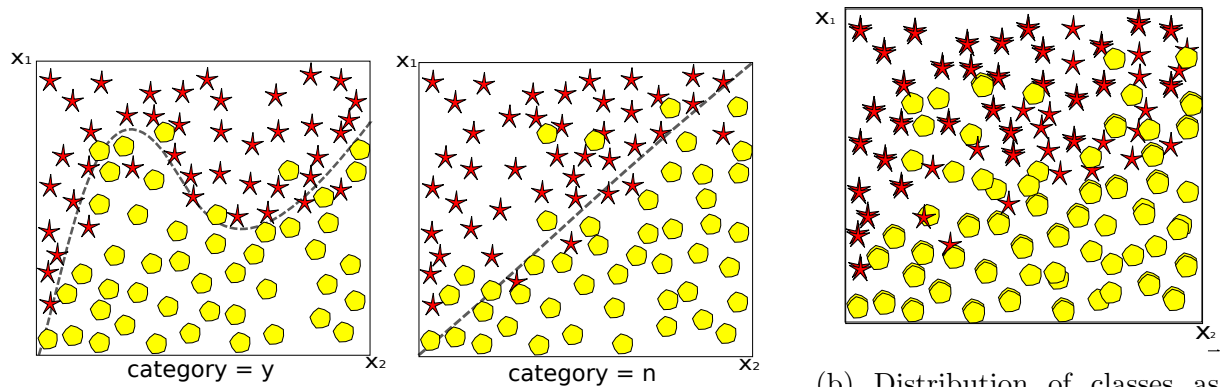
Decision trees offer a method to classify mixed datasets. A decision tree classifies an example by walking through the tree, taking branches that match with the current example, until a leaf is reached where a class label is assigned. Such paths can be seen as conjunctive logical expressions, where an example is classified by the leaf at the end of a path. Numeric data is supported by nodes which compare a numeric feature to a threshold, and branch based on if the feature value is greater or less than the threshold. This method however can only represent axis-parallel partitions within numeric space. Because of this limitation, decision trees can become quite complex by attempting to approximate a non-linear, or non-axis-parallel region using multiple axis parallel partitions of the feature space [17]. This drawback can lead to bad generalizability, and additionally, because during training, the amount of training data at each node decreases the further down the tree, the learnt boundary may not be very accurate.

Because the stepwise algorithm can represent more complex boundaries in decision trees, and decision trees are a natural fit for categorical data, merging the two algorithms together may be useful in facilitating the use of the stepwise algorithm on mixed datasets.

One method of merging the two algorithms together is a method we call Singly Augmented Dataset (SADS). The algorithm first utilizes the stepwise centroid algorithm to learn a classifier just based on the numeric features, and then the classification result returned by the classifier is treated as an extra feature of each example when constructing a decision tree with C4.5. This method treats the stepwise classifier as an ‘expert’ and the decision tree would utilize the classifier result if splitting based on the resulting classification causes an overall increase in purity for each created subset. This occurs when the stepwise classifier has developed a good separation between each class within numeric space.

The above method of merging the stepwise algorithm and decision trees has a possible downside if the categorical and numeric features are correlated. In Figure 3.4a, the distribution of classes within the numeric space is correlated to the value of the feature ‘category’. In this case, Figure 3.4b illustrates what the stepwise centroid classifier will be given to learn from in numeric space when ignoring the categorical features. The decision boundary in this figure is far less clear, so the accuracy of the stepwise centroid algorithm when finding a boundary will be lower. This may cause the feature to be ignored during the feature construction stage, if it is possible to branch on a single numeric feature and get better results once the categorical feature has been considered.

Because of this possible problem, we have developed another method of merging the two algorithms, called Multi Augmented Dataset (MADS). First we compute the cartesian product of all the possible values for every categorical feature plus a special value ‘?’ which indicates that the feature can take on any value. Then, for every generated tuple of feature values, we compute the subset of training data that matches the tuple. For each subset that has m or more examples, where m is some threshold controlling how large a subset must be, we train the stepwise centroid classifier over the numeric attributes present in the subset. Each classifier that is learnt is used as an additional feature when constructing a new decision tree. This should account for the possibility of categorical features having a correlation with the numeric features. However, the algorithm is brute force, and will not scale to mixed datasets with a more than a few categorical features, because of the exponential nature of the cartesian product calculation. Additionally, because the stepwise centroid classifier is trained on only a subset of training data, the subset may not be representative of the complete problem, and therefore the stepwise centroid classifier may be overfitting on the subset, which could confuse the greedy nature of the decision tree algorithm when performing feature selection.



(a) Two different distributions of numeric features ' x_1, x_2 ' correlated with a categorical feature 'category'

(b) Distribution of classes as seen by the stepwise centroid algorithm when just training on numeric features

Figure 3.4: Example classification where categorical features are correlated with numeric features.

4 Experiment Design

Four separate experiments have been designed in order to evaluate each algorithm described in the design section. The first experiment evaluates the GPU implementation of the designed PSO algorithm for classification. The second evaluates the effectiveness of the stepwise centroid classifier in terms of accuracy when compared to a single centroid classifier. The last two experiments evaluate the SADS and MADS algorithms respectively for their effectiveness on mixed datasets.

4.1 GPU implementation

The GPU implementation of both numeric classification algorithms, namely the stepwise centroid algorithm developed in this report, and the single centroid classification algorithm, are compared to an equivalent, sequential CPU implementation over a number of different data sets, in order to evaluate the speedup achieved by a GPU implementation of the algorithms. The following equation is used to calculate the speedup achieved from the GPU implementation:

$$\gamma = \frac{T_{\text{CPU}}}{T_{\text{GPU}}}$$

where T_{CPU} and T_{GPU} are the CPU and GPU times respectively.

Additionally, we also examine how the GPU and CPU implementations of the single centroid algorithm scale with the size of the classification problem (in terms of number of features, and number of examples), in order to understand how the effectiveness of the parallel GPU implementation. The single centroid algorithm was used over the stepwise algorithm because of the lack of problem dependence for running time. Additionally, the single centroid algorithm utilizes the main components of the GPU implementation that should be evaluated, such as the fitness function performance. To perform these experiments, we create a number of classification problems with random data, but fixed sizes. The number of features is initially fixed at 10, the number of classes is fixed at 2, and the number of examples is initially fixed at 1000. We then run the single centroid algorithm multiple times, increasing the number of examples by 100 each time up until 100000 examples. We record the time taken for each run, and then plot the results. We do the same

for the number of features, but we increase the amount by 10 each time, up until 1000 features. We record the times, and then plot the results.

4.2 Stepwise Centroid algorithm

To evaluate the performance of the developed stepwise centroid algorithm, the accuracy of the algorithm is compared to a nearest centroid PSO classification algorithm over multiple different real world datasets consisting of numeric features, with each dataset varying in the number of examples, features, and classes, along with varying in difficulty to classify. To evaluate if the difference between the stepwise centroid and single centroid algorithms is significant, a paired t -test with a p -value of 0.05 or less is utilized. The algorithm is ran over each data set 30 times, only modifying the random seed of the PSO algorithm on each run through.

Additionally, we have also considered run the designed algorithm on two synthetic data sets, in order to get a better understanding of the limitations of the representation, and how the algorithm performs for simple, easy to visualize boundaries.

4.3 MADS and SADS

To evaluate the augmented dataset approaches to classifying mixed data sets, the designed algorithms have been run over a number of different datasets, some including only numeric features, and others have a mixed number of each type of feature. For both experiments, each algorithm was executed 30 times, with the random seed given to PSO being the only variable changed on each run through. Then, for each experiment, the algorithm was compared to a normal decision tree in order to evaluate if a significant difference in classified accuracy was obtained, again using a paired t -test with a p -value of 0.05 or less in order to ensure the significance of the results.

4.4 Datasets

Table 4.1 gives an overview of the properties of each dataset used in the designed experiments. Each dataset has been partitioned into a training and test set, each with 2/3 of the data being used for training, and 1/3 being used for testing. Stratified sampling is used to partition the datasets. This method partitions the examples based on the class, and then samples from each partition individually to create the training and test set. For datasets with a large class imbalance, this helps ensure that the correct ratio of each class is present in both sets.

Table 4.1: Properties of all the datasets [2] used in our experiments

Dataset	Data size			Attribute Type			Classes
	Train	Test	Total	Categorical	Numeric	Total	
Australian Credit Approval	454	236	690	8	6	14	2
Credit Approval	430	223	653	9	6	15	2
Tae	99	52	151	3	1	4	3
Hepatitis	52	28	80	13	6	19	2
Heart	178	92	270	7	6	13	2
Abalone	2741	1436	4177	1	7	8	29
Haberman	201	105	306	0	3	3	2
Balance	416	209	625	0	4	4	3
Letter Recog	13200	6800	20000	0	16	16	26
Shuttle	28707	14806	43513	0	9	9	7
Iris	99	51	150	0	4	4	3
wdbc	374	195	569	0	30	30	2
Breast Cancer	450	233	683	0	9	9	2
Magic	12553	6467	19020	0	10	10	2
Banknote Authentication	905	467	1372	0	4	4	3
Skin Segmentation	161737	83320	245057	0	4	4	2

4.5 Configuration settings

The designed algorithms have many different parameters, and many external variables affect the outcome of the experiments. For each experiment we have utilized a NVIDIA GTX 780 GPU, and an Intel i5 2500k CPU. The parameters of the PSO algorithm are described in Table 4.2, where the values for w , c_1 , and c_2 have been chosen based on a paper by Eberhart *et al.* [6]. Additionally, the v_{\max} , v_{\min} and p_{\max} , p_{\min} values are used to confine each particles velocity and position respectively, to ensure particles stay within the search space.

The stepwise centroid algorithm also has the additional parameter n , which specifies how many iterations the algorithm will execute while the accuracy over the validation set has not increased. In our experiments, we have utilized the 10 for this value, which should allow the algorithm to add multiple centroids before accuracy increases, but not too high such that the algorithm ends up overfitting the validation set.

Additionally, the GPU implementation of the centroid algorithms utilizes a value of 1024 for the maximum number of threads in a block which increases the number of particle dimensions, or examples, depending on the kernel that are processed by each thread block.

The MADS algorithm has the additional parameter m controlling how large a subset of data must be before utilizing the stepwise centroid algorithm to learn a classifier for the

Table 4.2: Parameters of the PSO algorithm

Setting	Value
Number of Particles	500
Number of Iterations	100
Particle Topology	Ring
w	0.7968
c_1	1.4962
c_2	1.4962
v_{\max}	1.0
v_{\min}	0.0
p_{\max}	1.0
p_{\min}	0.0

subset of data. This has been set to 30, which allows the stepwise algorithm to train on a small amount of data, but may be necessary because of the sizes of each dataset.

5

Results and Discussion

5.1 GPU Implementation

Table 5.1: GPU and CPU times in seconds for both the stepwise centroid, and single centroid algorithms.

Data set	Stepwise Centroid			Single Centroid		
	GPU	CPU	Speedup	GPU	CPU	Speedup
Iris	0.8332	6.8957	8.28	0.0720	0.2137	30.86
Breast Cancer	1.5550	51.1669	32.90	0.0954	1.1767	12.33
wdbc	11.1900	137.4110	12.28	3.1090	0.2526	12.33
Banknote	1.4490	55.6737	38.42	0.0913	1.7202	18.84
Letter Recognition	804.3550	10125.2000	12.59	21.7417	696.4370	32.03
Shuttle	195.9270	5236.1800	26.73	6.9730	244.9380	35.13
Magic	75.6590	1620.7900	21.42	1.1908	36.7500	30.86
Skin Segmentation	214.8290	7152.6900	33.29	4.8352	182.1390	37.67
SPECTF	15.9339	49.7974	3.13	0.1974	1.1007	5.58

From Table 5.1, we can see that on most datasets, the GPU implementation is at least an order of magnitude faster than the sequential CPU implementation for both algorithms. However both the Iris and SPECTF datasets do not exhibit such a speed increase. Additionally, the speed increases differ drastically between datasets, and between algorithms on the same dataset.

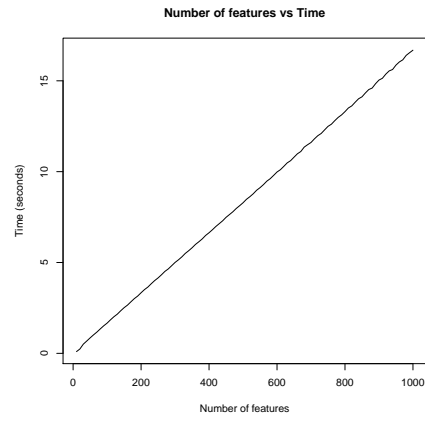
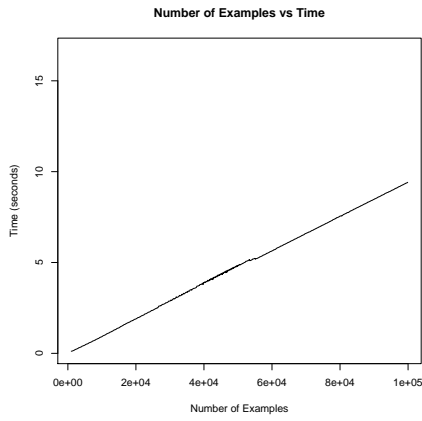
The reason why the stepwise centroid algorithm achieves different speed increases than the single centroid algorithm on the same dataset in many cases is because of the behaviour of the stepwise centroid algorithm being different. That is, the stepwise centroid algorithm iteratively adds new centroids to the classifier until the accuracy over the validation set has not increased for a number of iterations. Because of this, the runtime of the algorithm is output sensitive, since the time the algorithm takes is proportional to the amount of iterations the algorithm completes until a sufficient number of centroids is reached. The single centroid algorithm on the other hand does not have this property, only a single optimization is performed using PSO, and the runtime just depends on the size of the problem.

Speed increases differ between datasets for both algorithms however, and a small correlation can be seen between the size of each dataset and the speed increase achieved. The reason for these differences is because the problems differ in size, and difficulty, so therefore the speed increases can vary. However, we see that both the Iris dataset, and the SPECTF dataset fail to exhibit as high of a speed increase as the other datasets tested. The cause of this is that the number of examples in each of these datasets is very small. The GPU calculation of the fitness function explains why these differences occur. That is, because the fitness function is parallelized in such a way that more thread blocks are used as the training set grows, increasing the number of training examples increases the parallelism exhibited by the GPU, therefore giving larger speedups when compared to a sequential CPU implementation.

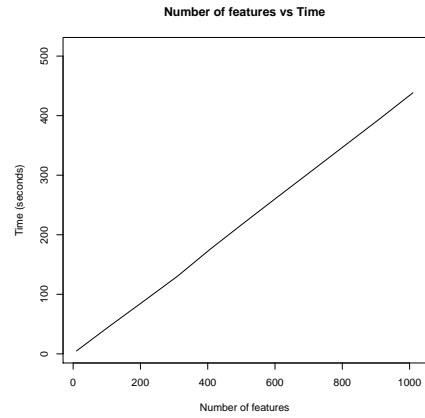
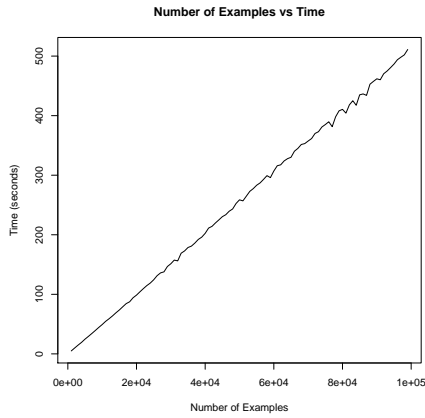
The letter recognition dataset has a large number of examples, but does not exhibit as large of a speedup as other datasets on the stepwise centroid algorithm. This is because of the number of classes in the problem. That is, many new centroids must be added on each iteration in order to properly learn all the decision boundaries between each class. This means more PSO optimizations must be performed, and more work done by the CPU overall due to the larger number of iterations. Additionally, more centroids means more memory reads to find the closest centroid to classify an example in the fitness function, and because memory can be slow to read, this increases the latency of the GPU implementation.

Figure 5.1 describes how the single centroid GPU implementation, which can be seen as a component of the stepwise centroid algorithm, scales with the size of the classification problem. That is, with the number of features and examples in the problem. The diagrams indicate that the time taken scales linearly with both the number of examples, and the number of features in the problem. This makes sense, because parallelization typically does not change the time complexity of an algorithm. The different scales on the x-axis should be noted. Because the number of features, and the number of training examples are different quantities of a problem, and the range of values tested for each parameter differ wildly. The GPU implementation appears to scale better with the number of examples in the training set, whereas increasing the number of features has a much larger effect on the running time of the algorithms. The reason why the algorithm scales better with an increasing number of training examples is because of how the fitness function is parallelized—increasing the number of examples increases the number of thread blocks used by the GPU algorithm. Therefore increasing the amount of work spread across the GPU multiprocessors. However, the reason why increasing the number of features has such a large effect on the speed of the algorithm is because of how high latency GPU memory access can be. That is, increasing the number of features increases the amount of reads of memory the GPU must perform when computing the closest centroid, or updating the positions. Therefore the speed of the algorithm overall is affected.

From the CPU diagrams, we can observe different behaviour in terms of how the algorithm scales with the problem size. Ignoring the order of magnitude of difference in time



(a) GPU Number of examples vs time (seconds) (b) GPU Number of features vs time (seconds)



(c) CPU Number of examples vs time (seconds) (d) CPU Number of features vs time (seconds)

Figure 5.1: Diagrams describing how the GPU implementation scales with different properties of a problem

between the CPU and GPU implementations, we see that the CPU algorithm scaling of time between the number of features and number of examples is about the same. This can be seen by considering the initial values used to generate these graphs. That is, when the number of examples is varying, the number of features is fixed to 10, this means that when there are 100000 examples, the time complexity of the algorithm is bounded above by $|C| \times 100000 \times 10 = 10^6|C|$. On the other hand, consider that when varying the number of features, the number of examples is fixed to 1000, and when there are 1000 features, the time complexity of the algorithm is bounded above by $|C| \times 1000 \times 1000 = 10^6|C|$. That is, the amount of work in both cases is around equal, and given that the graphs are very similar, we can conclude that for the CPU implementation, increasing the input size in any dimension would equally increase the computation time.

5.2 Stepwise Centroid Accuracy

Table 5.2: Average accuracies of the stepwise centroid classifier and the single centroid classifier. With accuracy on training in brackets.

Data set	Stepwise Centroid Classifier	Single Centroid Classifier
Iris	0.9163 (0.9808)	0.9294* (1.0000)
Banknote	0.9950* (0.9992)	0.9866 (0.9969)
Letter Recog	0.3838* (0.3885)	0.1923 (0.1920)
Shuttle	0.9955* (0.9958)	0.9714 (0.9713)
Magic	0.8313* (0.8399)	0.7908 (0.8045)
Skin Segment	0.9940* (0.9940)	0.9476 (0.9483)
Breast Cancer	0.9624 (0.9796)	0.9654 (0.9811)
wdbc	0.9636 (0.9846)	0.9685 (0.9841)
SPECTF	0.7589 (0.8988)	0.7686 (0.9192)

Table 5.2 displays the accuracies of both the stepwise centroid, and single centroid PSO classification algorithms over multiple datasets. The average accuracy over the training data is shown in brackets, and boldface is used to highlight the best result for each data set, where the presence or absence of * indicates whether or not the result is statistically significant. The results indicate that the developed stepwise centroid algorithm has had some success on the chosen data sets, achieving the same, or better results on all except one dataset.

Because of the size of the Iris dataset, the amount of data used for training within the stepwise centroid algorithm is quite small, and the algorithm further partitions the dataset into a separate validation set. Additionally, given that the single centroid algorithm achieved a higher training accuracy than the stepwise algorithm, this is indicative of the partitioning of the training data into a validation set meant the stepwise algorithm lacked enough data required to correctly learn a decision boundary for the problem, despite the ability to add new centroids.

Additionally, the results show that the stepwise centroid algorithm failed to achieve an increase in accuracy on 3 of the tested datasets. Many possible reasons exist for why this could happen. If a dataset has a simple boundary, but a large amount of noise, then increasing the power of the classifier cannot further increase the accuracy of a model. Additionally, if the dataset is small, there may not be enough data in the training set to recognise the correct boundary, and therefore the algorithm will be unable to further increase the accuracy.

For further understanding of the developed algorithm, we can consider the performance over a number of synthetic datasets. Figure 5.2 displays two non-linear functions, namely a circle and a quadratic function, with the decision boundaries learnt by the stepwise centroid algorithm. These diagrams show that the boundaries learnt do not exactly match the desired function. This is caused by the centroids defining linear boundaries, with multiple centroids approximating curved surfaces with multiple centroids. This means the boundaries may not be exact, since the boundaries only have to match a finite amount of training data. On each dataset, the stepwise algorithm achieved 95.5% and 98.6% accuracy respectively.

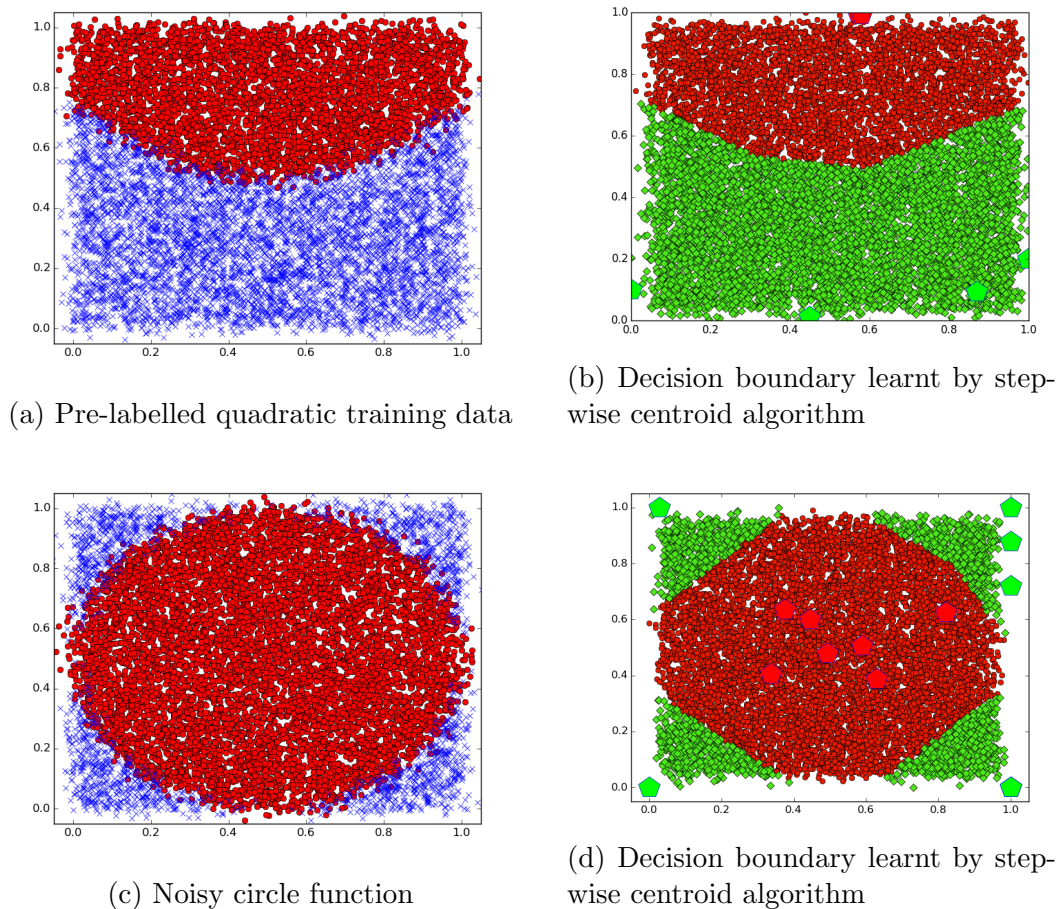
5.2.1 Mixed datasets

Table 5.3: Results achieved by the SADS algorithm

Data set	Test Accuracy		Number of Leaves	
	Decision Tree	SADS	Decision Tree	SADS
Australian Credit Approval	86.86	86.82	35	33.2
Shuttle	99.93	99.92	20	25.1
Tae	51.92*	49.93	16	10.7
Letter Recog	86.19*	85.15	993	1215.5
Haberman	73.33	76.52*	1	2
wdbc	95.38	96.52*	10	4.6
Breast Cancer	92.70	96.36*	6	2.1
Magic	84.07	85.29*	268	210.0
Balance	78.87	86.47*	27	6.5
Iris	90.20	91.48*	4	3.2
Credit Approval	86.10*	84.41	21	22.9
Hepatitis	89.29*	83.50	4	2.4
Heart	77.17	78.26	11	7.0
Abalone	22.77*	21.88	783	1114.6

Table 5.3 summarises the results of the SADS algorithm on multiple different datasets, each with a different number of categorical features. The results show that the algorithm only achieves better results on the purely numeric datasets, and manages to get worse

Figure 5.2: Artificial classification problems and decision boundaries learnt for them



results on some of the datasets that include categorical features. Additionally, it appears that on a single purely numeric dataset, namely the letter recognition dataset, that the non augmented decision tree achieves better results. In addition to the accuracy results over the testing data, the table also describes the average number of leaves in each tree constructed by the algorithm, which gives us an idea of the complexity of the learnt model. This data indicates that on a few datasets, mainly the datasets that so no improvement, the complexity of the learnt decision tree is higher.

As described in the design section a likely reason for this bad result, is a high correlation between categorical features and the numeric features. If this is the case, the stepwise centroid algorithm will not have enough information about the problem to achieve a higher classification accuracy when just given the numeric features.

The result on letter recognition is also notable, and is most likely attributable to a possible shortcoming in the stepwise learning algorithm. If we compare the results, in a statistically

invalid way, to the result achieved by the stepwise centroid algorithm itself, the results that the accuracy achieved by the decision tree far outweighs the accuracy achieved by the stepwise algorithm. This is most likely caused by the number of classes in the letter recognition dataset, and the way the stepwise centroid algorithm adds new centroids to a classifier. That is, a random class is assigned the new centroid added to the classifier at each iteration of the algorithm, and a large number of classes in the problem decreases the likelihood of the same class being chosen multiple times in a row, even with the algorithm continuing to add new centroids once the accuracy hasn't increased in the last iteration. This can be a problem if defining a well-defined boundary in the feature space requires multiple centroids of the same class, for example if classes are separated by a complex curved surface. On the other hand while decision trees can suffer from generalizability issues because of the way numeric features are handled, the amount of classes only affects how the purity of a given split is calculated, but does not introduce a large bias to how the decision tree is constructed.

The average number of leaves in the decision tree constructed by the SADS algorithm for letter recognition dataset is also interesting. That is, even though it is unlikely that the feature created by stepwise centroid algorithm introduces any additional information to the decision tree, the generated tree still appears to be more complex. The reason for this is likely due to the stepwise algorithm creating a feature which is marginally better than any single feature in terms of partitioning the training data by class. This makes the constructed feature look superior to the other features when considered as a feature to split on during the decision tree construction. This can lead to the replication problem [27], where the decision tree constructs equivalent subtrees in multiple different branches of a tree, because a split higher up in the tree partitioned the data such that the data in some partitions could be split in the same way.

Table 5.4 summarises the results of the MADS algorithm on multiple different data sets, including numeric data sets. The results are similar to the SADS algorithm, but show relatively large variation on some data sets when compared to the SADS algorithm. The algorithm appears to achieve a better result on one categorical data set, but appears to be worse on three other datasets.

The MADS algorithm builds a decision trees over a dataset with the original numeric features, and multiple stepwise centroid features. The number of leaves built for each tree, for most datasets, indicates that the added features are being utilized during the construction of the decision tree, since if the features are ignored, the built tree will be equivalent to the one built by the original decision tree algorithm.

The performance of the MADS algorithm could be because of overfitting the data in each subset, or there existing only a small amount of training data in each problem. That is, If the stepwise algorithm is overfitting on the subsets of the training data, then the generalisability of the built tree will be negatively affected. This is because the learnt

Table 5.4: Results achieved by the MADS algorithm

Data set	Test Accuracy		Number of Leaves	
	Decision Tree	MADS	Decision Tree	MADS
Australian Credit Approval	86.86*	84.47	35	29.5
Shuttle	99.93*	99.91	20	24.0
Tae	51.92	52.72	16	12.9
Letter Recog	86.19*	85.03	993	1220.8
Haberman	73.33	76.88*	1	2
wdbc	95.38	96.41*	10	4.4
Breast Cancer	92.70	96.17*	6	2.0
Magic	84.07	85.17*	268	210.1
Balance	78.87	87.39*	27	5.9
Iris	90.20	91.95*	4	3.1
Credit Approval	86.10*	81.10	21	21.2
Hepatitis	89.29*	84.48	4	2.5
Heart	77.17	80.06*	11	5.7
Abalone	22.77*	21.28	783	2064.5

features will appear to be good to the decision tree construction algorithm, but will not generalise well to unseen data. Additionally, if the amount of data in a subset is small, then the learnt model by the stepwise centroid algorithm will not necessarily be representative, but will fit the training data well. This will cause the decision tree construction algorithm to use the constructed feature, but the generalizability of the learnt tree will be affected negatively.

We also see an increase in accuracy on one categorical dataset, namely the heart dataset. This could be explained by a number of reasons. Decision trees are constructed using a greedy algorithm, where at each node in the tree a feature is chosen as a test based on some local purity measure, where gain ratio is the most common choice. The greedy nature of this algorithm means that slight variations in what features are selected may have a drastic effect on the built tree, and the accuracy of the built tree. That is, it is not immediately possible to conclude based on the result that the difference is because the additional features provide additional information to the decision tree construction algorithm, since the decision tree can be quite sensitive to variations in what features are chosen.

6

Conclusion

We have developed a new PSO based classification algorithm, using centroid based classifier representation. Previous research using PSO for training a centroid based classifier utilized a single centroid per class. We extended this representation, and therefore the learning algorithm to allow for multiple centroids per class, which means more complex decision boundaries may be represented. The choice of classifier representation was decided because the continuous nature was a natural fit for PSO, a continuous optimization algorithm. Results on a few real world datasets showed promising results when compared to the single centroid algorithm, however by running the algorithm on two synthetic datasets, the diagrams showed the learnt decision boundaries may not fully represent the intended boundary, because of the developed algorithm representing piece-wise linear decision boundaries.

Along with this, the developed classification algorithm was extended to mixed datasets by utilizing the developed algorithm to construct new features for each example based on the existing numeric features. Then, a standard decision tree construction algorithm was used to build a tree given a training set with the additional features. Two different approaches to achieving this were evaluated, SADS, which developed a single feature over all the numeric features, and MADS, which created a number of features, each for a different subset corresponding to some conjunction of feature values. We observed from the limited experiments conducted to evaluate the developed methods for mixed data classification that both the SADS or MADS algorithms were not effective for mixed data sets, and did not make a significant difference to the standard decision tree algorithm on the real world datasets that were used for testing.

Finally, we also evaluated the effectiveness of a GPU implementation of the PSO algorithm for classification, and investigated how the algorithm scales with the size of the classification problem. Results showed that the GPU algorithm was at least an order of magnitude faster than an equivalent, sequential CPU implementation on most datasets tested. However, for two small datasets, the speedup achieved was not as large, because of the GPU not being given enough work to parallelize effectively. Additionally, in terms of scaling with the size of the data, we see that the single centroid algorithm scales both linearly in the number of features, and the amount of training examples, which is expected because of the time complexity of the algorithm. However, although the dimensions of

the problem are not directly comparable quantities, we saw that the time scales better with an increasing number of examples than with the number of features. This indicates that the algorithm scales better with the number of examples than with the number of features. When compared to the CPU implementation, this effect is particularly noticable. That is, the CPU implementation scales the same in both dimensions, but for the GPU implementation, we see that increasing the number of examples causes the algorithm to be more efficient.

6.1 Limitations and Future Work

In this research, we compared a GPU implementation of an algorithm to a sequential, CPU implementation. This is not an entirely sufficient comparison [15] because the full capabilities of the CPU have not been fully taken advantage of. Additionally, the exact amount of utilization of the GPU has not been explicitly investigated. Therefore the comparison is only useful as an indicator of the relative performance of the GPU. Future work will focus on further optimising a CPU implementation by taking advantage of the SIMD instructions offered by Intel, and the various parallelization strategies on the CPU. Additionally, we will also do further investigation into exactly how the GPU's resources are utilized by our algorithm implementation.

The designed multi centroid classification algorithm can suffer from the inability to correctly define curved linear surfaces, due to the piecewise linear nature of the boundaries the centroids define. Future work could focus on extending the designed algorithm to use different kernels [1], or a fuzzy set method [13] in order allow for curved surfaces to be defined. However, doing so may limit the ability of the algorithm to represent non-curved, but discontinuous boundaries.

The datasets used to evaluate the designed approaches, MADS, and SADS, were quite small in the number of examples used. This was an issue when evaluating why the developed algorithms were not effective on mixed data sets. Future work could focus on utilising larger, but still feasible (not too many categorical feature) categorical datasets in order to get a better idea of the behaviour of the designed algorithm.

Bibliography

- [1] A Aizerman, Emmanuel M Braverman, and LI Rozoner. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and remote control*, 25:821–837, 1964.
- [2] K. Bache and M. Lichman. UCI machine learning repository, 2013.
- [3] Altaf QH Badar, BS Umre, and AS Junghare. Reactive power control using dynamic particle swarm optimization for real power loss minimization. *International Journal of Electrical Power & Energy Systems*, 41(1):133–136, 2012.
- [4] Niccolo’ Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. infant: Nfa pattern matching on gpgpu devices. *SIGCOMM Comput. Commun. Rev.*, 40(5):20–26, October 2010.
- [5] I. De Falco, a. Della Cioppa, and E. Tarantino. Facing classification problems with Particle Swarm Optimization. *Applied Soft Computing*, 7(3):652–658, June 2007.
- [6] Russ C Eberhart and Yuhui Shi. Comparing inertia weights and constriction factors in particle swarm optimization. In *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, volume 1, pages 84–88. IEEE, 2000.
- [7] Russell C Eberhart and Xiaohui Hu. Human tremor analysis using particle swarm optimization. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 3. IEEE, 1999.
- [8] Venu G Gudise and Ganesh K Venayagamoorthy. Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks. In *Swarm Intelligence Symposium, 2003. SIS’03. Proceedings of the 2003 IEEE*, pages 110–117. IEEE, 2003.
- [9] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.
- [10] Chia-Feng Juang. A hybrid of genetic algorithm and particle swarm optimization for recurrent network design. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(2):997–1006, 2004.
- [11] J Kennedy and R Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948, 1995.

- [12] J. Kennedy and R. Mendes. Population structure and particle swarm performance. In *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, volume 2, pages 1671–1676, 2002.
- [13] George Klir and Bo Yuan. *Fuzzy sets and fuzzy logic*, volume 4. Prentice Hall New Jersey, 1995.
- [14] Yossi Kreinin. SIMD < SIMT < SMT: parallelism in NVIDIA GPUs. yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html. Accessed Oct 6, 2014.
- [15] Pavel Kromer, Jan Platos, and Vaclav Snasel. A brief survey of advances in Particle Swarm Optimization on Graphic Processing Units. *Nature and Biologically Inspired Computing*, pages 82–88, 2013.
- [16] A.W. Mohemmed and Mengjie Zhang. Evaluation of particle swarm optimization based centroid classifier with different distance metrics. In *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 2929–2932, June 2008.
- [17] Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. A system for induction of oblique decision trees. *arXiv preprint cs/9408103*, 1994.
- [18] Luca Mussi, Youssef S.G. Nashed, and Stefano Cagnoni. GPU-based asynchronous particle swarm optimization. *Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO '11*, page 1555, 2011.
- [19] C Nvidia. NVIDIA CUDA C Programming Guide. *Changes*, page 173, 2011.
- [20] Riccardo Poli. Analysis of the Publications on the Applications of Particle Swarm Optimisation. *Journal of Artificial Evolution and Applications*, 2008(2):1–10, 2008.
- [21] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, August 2007.
- [22] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
- [23] Tiago Sousa, Arlindo Silva, and Ana Neves. Particle Swarm based Data Mining Algorithms for classification tasks. *Parallel Computing*, 30(5-6):767–783, May 2004.
- [24] B Sowmya and MP Sunil. Minimization of floorplanning area and wire length interconnection using particle swarm optimization. *International Journal of Emerging Technology and Advanced Engineering Volume*, 3, 2013.
- [25] Ponnuthurai N Suganthan. Particle swarm optimiser with neighbourhood operator. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 3. IEEE, 1999.
- [26] Cole Trapnell and Michael C. Schatz. Optimizing data intensive {GPGPU} computations for {DNA} sequence alignment. *Parallel Computing*, 35(89):429 – 440, 2009.

-
- [27] Der-Shung Yang, Gunnar Blix, and Larry A. Rendell. The replication problem: A constructive induction approach. In Yves Kodratoff, editor, *Machine Learning EWSL-91*, volume 482 of *Lecture Notes in Computer Science*, pages 44–61. Springer Berlin Heidelberg, 1991.
 - [28] You Zhou and Ying Tan. GPU-based parallel particle swarm optimization. *2009 IEEE Congress on Evolutionary Computation*, 2009.